CS:5810 Formal Methods in Software Engineering

Reactive Systems and the Lustre Language

---

Copyright 2015-18, Adrien Champion and Cesare Tinelli, the University of Iowa. These notes are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holder.
Embedded systems development

Pivot language between design and code should have clear and precise semantics, and be consistent with design / prototype formats and target platforms.
Embedded systems development

Pivot language between design and code should
• have clear and precise semantics, and
Embedded systems development

Pivot language between design and code should

- have clear and precise semantics, and
- be consistent with design / prototype formats and target platforms
Lustre: a synchronous dataflow language

- **Synchronous:**
  
a base clock regulates computations;
  computations are inherently parallel

- **Dataflow:**
  
  inputs, outputs, variables, constants . . . are endless streams of values
Lustre: a synchronous dataflow language

- **Synchronous:**
  a base clock regulates computations; computations are inherently parallel

- **Dataflow:**
  inputs, outputs, variables, constants . . . are endless streams of values

- **Declarative:**
  set of equations, no statements
Lustre: a synchronous dataflow language

- **Synchronous:**
  a base clock regulates computations;
  computations are inherently parallel

- **Dataflow:**
  inputs, outputs, variables, constants . . . are endless streams of values

- **Declarative:**
  set of equations, no statements

- **Reactive systems:**
  Lustre programs run forever
  At each clock tick they
  - compute outputs from their inputs
  - before the next clock tick
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
```

tel

Circuit view:

```
+  /  
|  |  |
x  y  2.0
    → out
```

4 / 17
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
 tel

Mathematical view:

∀i ∈ ℕ, out_i = \frac{x_i + y_i}{2}
```
A simple example

node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
end

Transition system unrolled view:

clock ticks 0 1 2 3 ...

6/17
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
tel
```

Transition system unrolled view:

```
<table>
<thead>
<tr>
<th>x0</th>
<th>y0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x0 + y0</td>
<td>2.0</td>
</tr>
<tr>
<td>out0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x1</th>
<th>y1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x1 + y1</td>
<td>2.0</td>
</tr>
<tr>
<td>out1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x2</th>
<th>y2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x2 + y2</td>
<td>2.0</td>
</tr>
<tr>
<td>out2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x3</th>
<th>y3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>x3 + y3</td>
<td>2.0</td>
</tr>
<tr>
<td>out3</td>
<td></td>
</tr>
</tbody>
</table>

clock ticks 0 1 2 3 ...
```
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
end
```

Transition system unrolled view:

```
<table>
<thead>
<tr>
<th>Clock ticks</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4.0</td>
<td>0.0</td>
<td>1.0</td>
<td>7.0</td>
<td>1.0</td>
</tr>
<tr>
<td>y</td>
<td>6.0</td>
<td>7.0</td>
<td>1.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
<tr>
<td>out</td>
<td>5.0</td>
<td>3.5</td>
<td>1.0</td>
<td>4.0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
```
Combinational programs

- Basic types: bool, int, real

- Constants (i.e., constant streams):
  
  \[
  \begin{array}{c|cccccccc}
  2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & \ldots \\
  \text{true} & \text{true} & \text{true} & \text{true} & \text{true} & \text{true} & \text{true} & \text{true} & \ldots \\
  \end{array}
  \]
Combinational programs

- Basic types: bool, int, real

- Constants (i.e., constant streams):
  
  | 2  | 2  | 2  | 2  | 2  | 2  | ... 
  |-----|----|----|----|----|----|------|
  | true | true | true | true | true | true | true | ...

- Pointwise operators:

  | x  | x₀ | x₁ | x₂ | x₃ | x₄ | ... 
  |-----|----|----|----|----|----|------|
  | y  | y₀ | y₁ | y₂ | y₃ | y₄ | ... 
  | x + y | x₀ + y₀ | x₁ + y₁ | x₂ + y₂ | x₃ + y₃ | x₄ + y₄ | ... 

- All classical operators are provided
Conditional expressions:

\[
\text{node max (n1,n2: real) returns (out: real);} \\
\text{let} \\
\quad \text{out = if (n1 >= n2) then n1 else n2;} \\
\text{tel}
\]

- Functional “if ... then ... else ...”
- It is an expression, not a statement
Combinational programs

Conditional expressions:

```plaintext
node max (n1,n2: real) returns (out: real);
let
    out = if (n1 >= n2) then n1 else n2;
 tel
```

- Functional “if ... then ... else ...”
- It is an expression, **not a statement**
  ```plaintext
  -- This does not compile
  if (a >= b) then m = a else m = b;
  ```
Combinational programs

Local variables:

```plaintext
define max (a, b: real) returns (out: real);
var
    condition: bool;
let
    out = if condition then a else b;
    condition = a >= b;
tel
```
Local variables:

```plaintext
node max (a, b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
tel
```

- Order does not matter
- Set of equations not sequence of statements
Combinational programs

Local variables:

```plaintext
node max (a, b: real) returns (out: real);
var
    condition: bool;
let
    out = if condition then a else b;
    condition = a >= b;
tel
```

- Order does not matter
- Set of equations not sequence of statements
- Causality is resolved syntactically
Combinational programs

Combinational recursion is forbidden:

\[ x = 1 / (2 - x); \]
Combinational programs

Combinational recursion is forbidden:

\[ x = 1 \div (2 - x); \]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step
Combinational programs

Combinational recursion is forbidden:

\[ x = \frac{1}{2 - x}; \]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step

Syntactic loop:

\[
\begin{align*}
  x &= \text{if } c \text{ then } y \text{ else } 0; \\
  y &= \text{if } c \text{ then } 1 \text{ else } x;
\end{align*}
\]
Combinational programs

Combinational recursion is forbidden:

\[ x = \frac{1}{2 - x}; \]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step

Syntactic loop:

\[ x = \text{if } c \text{ then } y \text{ else } 0; \]
\[ y = \text{if } c \text{ then } 1 \text{ else } x; \]

- not a real (semantic) loop:
  \[ x = \text{if } c \text{ then } 1 \text{ else } 0; \]
  \[ y = x; \]
- but still forbidden by Lustre
Previous operator \( \text{pre} \):

\[
(\text{pre } x)_0 \quad \text{is undefined (nil)}
\]

\[
(\text{pre } x)_i = x_{i-1} \quad \text{for } i > 0
\]
Memory programs

Previous operator $\text{pre}$:
- $(\text{pre } x)_0$ is undefined (nil)
- $(\text{pre } x)_i = x_{i-1}$ for $i > 0$

Initialization $\rightarrow$:
- $(x \rightarrow y)_0 = x_0$
- $(x \rightarrow y)_i = y_i$ for $i > 0$
Memory programs

Previous operator \( \text{pre} : \)

\[
\begin{align*}
(\text{pre } x)_0 & \quad \text{is undefined ( nil )} \\
(\text{pre } x)_i & = x_{i-1} \quad \text{for } i > 0
\end{align*}
\]

Initialization \( \rightarrow : \)

\[
\begin{align*}
(x \rightarrow y)_0 & = x_0 \\
(x \rightarrow y)_i & = y_i \quad \text{for } i > 0
\end{align*}
\]

Examples:

\[
\begin{array}{c|cccccccc}
\text{pre } x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
\hline
x & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
\]
Memory programs

Previous operator \( \text{pre} \): 
\[
(\text{pre} \ x)_0 \quad \text{is undefined (nil)} \\
(\text{pre} \ x)_i = x_{i-1} \quad \text{for } i > 0
\]

Initialization \( \rightarrow \): 
\[
(x \rightarrow y)_0 = x_0 \\
(x \rightarrow y)_i = y_i \quad \text{for } i > 0
\]

Examples:

\[
\begin{array}{c|cccccc}
\text{x} & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
\text{pre} \ x & // & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots
\end{array}
\]
Memory programs

Previous operator \texttt{pre}:
\[(\texttt{pre } x)_0\] is undefined (\texttt{nil})
\[(\texttt{pre } x)_i = x_{i-1}\] for \(i > 0\)

Initialization \texttt{s->}:
\[(x \rightarrow y)_0 = x_0\]
\[(x \rightarrow y)_i = y_i\] for \(i > 0\)

Examples:

\begin{tabular}{c|cccccccc}
  \(x\) & \(x_0\) & \(x_1\) & \(x_2\) & \(x_3\) & \(x_4\) & \(x_5\) & \ldots \\
  \texttt{pre } x & // & \(x_0\) & \(x_1\) & \(x_2\) & \(x_3\) & \(x_4\) & \ldots \\
  \(y\) & \(y_0\) & \(y_1\) & \(y_2\) & \(y_3\) & \(y_4\) & \(y_5\) & \ldots \\
  \(x \rightarrow y\) & & & & & & & \\
\end{tabular}
Memory programs

Previous operator \texttt{pre}:
\[
\begin{align*}
\texttt{(pre } x)_{0} & \quad \text{is undefined (nil)} \\
\texttt{(pre } x)_{i} = x_{i-1} & \quad \text{for } i > 0
\end{align*}
\]

Initialization \texttt{->}:
\[
\begin{align*}
\texttt{(x } \rightarrow y)_{0} & = x_{0} \\
\texttt{(x } \rightarrow y)_{i} = y_{i} & \quad \text{for } i > 0
\end{align*}
\]

Examples:
\[
\begin{array}{c|cccccccc}
\text{x} & x_{0} & x_{1} & x_{2} & x_{3} & x_{4} & x_{5} & \ldots \\
\text{pre x} & \text{//} & x_{0} & x_{1} & x_{2} & x_{3} & x_{4} & \ldots \\
\text{y} & y_{0} & y_{1} & y_{2} & y_{3} & y_{4} & y_{5} & \ldots \\
\text{x } \rightarrow y & x_{0} & y_{1} & y_{2} & y_{3} & y_{4} & y_{5} & \ldots \\
\end{array}
\]
Memory programs

Previous operator \( \text{pre} : \)

\[(\text{pre } x)_0 \text{ is undefined ( } \text{nil} \text{)}\]
\[(\text{pre } x)_i = x_{i-1} \text{ for } i > 0\]

Initialization \( \rightarrow : \)

\[(x \rightarrow y)_0 = x_0\]
\[(x \rightarrow y)_i = y_i \text{ for } i > 0\]

Examples:

\[
\begin{array}{c|cccccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & // & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  y & & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  x \rightarrow y & & x_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  2 & & 2 & 2 & 2 & 2 & 2 & \ldots \\
  2 \rightarrow (\text{pre } x) & & \ldots 
\end{array}
\]
Memory programs

Previous operator \( \text{pre} : \)

\[(\text{pre } x)_0 \text{ is undefined (nil)} \]
\[(\text{pre } x)_i = x_{i-1} \text{ for } i > 0 \]

Initialization \( \rightarrow : \)

\[(x \rightarrow y)_0 = x_0 \]
\[(x \rightarrow y)_i = y_i \text{ for } i > 0 \]

Examples:

\[
\begin{array}{c|cccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & // & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  y & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  x \rightarrow y & x_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  2 & 2 & 2 & 2 & 2 & 2 & 2 & \ldots \\
  2 \rightarrow (\text{pre } x) & 2 & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
\end{array}
\]
Memory programs

Recursive definition using \texttt{pre}:

\[
\begin{align*}
n = 0 & \rightarrow 1 + \texttt{pre}\ n; \\
a = \texttt{false} & \rightarrow \texttt{not}\ \texttt{pre}\ a;
\end{align*}
\]

\[
\begin{array}{c|c}
n & 0 \\
a & \texttt{false}
\end{array}
\]
Recursive definition using \texttt{pre}:

\begin{align*}
n &= 0 \rightarrow 1 + \texttt{pre} \ n; \\
\texttt{a} &= \texttt{false} \rightarrow \texttt{not pre} \ \texttt{a};
\end{align*}

\begin{tabular}{c|cccccc}
\texttt{n} & 0 & 1 & 2 & 3 & \ldots \\
\texttt{a} & \texttt{false}
\end{tabular}
Recursive definition using \texttt{pre}:

\[
\begin{align*}
\text{n} & = 0 \rightarrow 1 + \text{pre n}; \\
\text{a} & = \text{false} \rightarrow \text{not pre a};
\end{align*}
\]

\[
\begin{array}{c|cccccc}
\text{n} & 0 & 1 & 2 & 3 & \ldots \\
\text{a} & \text{false} & \text{true} & \text{false} & \text{false} & \text{true} & \ldots \\
\end{array}
\]
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel

| signal | 0 1 1 0 1 0 0 ...
|--------|
| e      | 1 0 0 1 0 1 1 ...
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
</tr>
</tbody>
</table>
node guess (signal: bool) returns (e: bool);
let
e = false -> signal and not pre signal;
tel

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 0 1 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0 1 0 0 1 0 0 1 0 ...</td>
</tr>
</tbody>
</table>
Memory programs: examples

Raising edge:

```plaintext
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
```

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0 1 0 0 1 0 ...</td>
</tr>
</tbody>
</table>
Memory programs: examples

```plaintext
node guess (n: int) returns (out1, out2: int);
let
    out1 = n -> if (n < pre out1) then n else pre out1;
    out2 = n -> if (n > pre out2) then n else pre out2;
```

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td></td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
    out1 = n -> if (n < pre out1) then n else pre out1;
    out2 = n -> if (n > pre out2) then n else pre out2;
tel

| n    | 4  2  3  0  3  7   ...
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4 2 2 0 0 0 0 ...</td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4 2 2 0 0 0 0 ...</td>
</tr>
<tr>
<td>out2</td>
<td>4 4 4 4 4 7 ...</td>
</tr>
</tbody>
</table>
Min and max of a sequence:

```plaintext
node guess (n: int) returns (out1, out2: int);
let
    out1 = n -> if (n < pre out1) then n else pre out1;
    out2 = n -> if (n > pre out2) then n else pre out2;
```

tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4 2 2 0 0 0 0 ...</td>
</tr>
<tr>
<td>out2</td>
<td>4 4 4 4 4 7 7 ...</td>
</tr>
</tbody>
</table>
Design a node

```python
node switch (on, off: bool) returns (state: bool);
```

such that:

- state raises (false to true) if on;
- state falls (true to false) if off;
Design a node

\[
\text{node switch (on, off: bool) returns (state: bool)};
\]

such that:

- state raises (false to true) if on;
- state falls (true to false) if off;
- everything behaves as if state was false at the origin;
- switch must work properly even if on and off are the same.
Compute the sequence 1, 1, 2, 3, 5, 8 \ldots
Compute the sequence 1, 1, 2, 3, 5, 8, 13, 21 \ldots

Fibonacci sequence:

\[ u_0 = u_1 = 1 \]
\[ u_n = u_{n-1} + u_{n-2} \quad \text{for } n \geq 2 \]
These notes are based on the following lectures notes:

The Lustre Language — Synchronous Programming
by Pascal Raymond and Nicolas Halbwachs
Verimag-CNRS